Getting Started with IBM Bluemix

Hands-On Workshop

Exercise 5c: Test-driven development

Test-driven development

Implement the divisibleBy function

To implement the divisibleBy funtion, you generate the first test that the implementation must pass. Solving the problem requires a function to discover whether one number is divisible by another number, so the first test will check whether 3 is divisible by 3.

Before you write the test, you'll add another test framework: Chai. Chai is a behavior-driven development (BDD) and test-driven development (TDD) assertion library for Node.js and a browser that can be paired with any JavaScript testing framework. Chai provides the test capability to say "I expect this to be true" or "this should be true."

Before you write a test, install Chai and then configure Mocha to use Chai:

1. Install and configure Chai:
   a. In the Terminals view, enter the following command:

      ```
      npm install chai --save-dev
      ```

   b. Select the project name in Project Explorer and press F5 to refresh the Eclipse project content.

   c. Create a new file in the test folder named `support.js` and add the following content:

      ```
      var chai = require("chai");
      global.expect = chai.expect;
      ```

   d. Create a new file in the test folder named `mocha.opts` and add the following content:

      ```
      --require test/support
      ```

   e. Save both files.

      The files configure Mocha to use the *expect* functionality from Chai.

2. Create the first test:
   a. Create a new file in the test folder named `fizzbuzz.test.js` by right-clicking the test folder in the Project Explorer view and then click **New > File.**

   b. Enter the following code to get access to the code that you are about to write to pass the test:

      ```
      var FizzBuzz = require("../fizzbuzz.js");

      describe("Fizzbuzz", function() {
        var f = new FizzBuzz();

        describe("divisibleBy()", function() {
      ```

```
    it("when divisible", function() {
      expect(f.divisibleBy(3, 3)).to.be.eql(true);
    });
  });
});
```

The first test is now complete, so you can write the code that's needed to pass the test.

Rather than having to manually run the tests, you can specify an option to continually run Mocha. Therefore, every time that a file is changed, the tests are automatically run. Use the `-w` command-line option to run tests continually.

3. Configure Mocha to continually run tests:
   a. In the Terminals view, run the following command:
      - **Mac and Linux**: `node_modules/.bin/mocha -w`
      - **Windows**: `node_modules\.bin\mocha -w`

      You will see that the tests are failing because of an error:
      `Error: Cannot find module '../fizzbuzz.js'`

   b. In the root of the project, create the file `fizzbuzz.js`.

   c. Add the following code to configure the `fizzbuzz.js` file:

```
var FizzBuzz = function (){
};

module.exports = FizzBuzz;
```

   d. Save the file. You should now see the tests run with the following result:

```
Fizzbuzz
    divisibleBy()
      1) when divisible

  jshint
    ✓ should pass for working directory (99ms)


  1 passing (114ms)
  1 failing

  1) Fizzbuzz divisibleBy() when divisible:
     TypeError: Object [object Object] has no method 'divisibleBy'
      at Context.<anonymous> (test/fizzbuzz.test.js:8:16)
```

Now, you can implement the divisibleBy function, but you should write only the minimal amount of code to pass the written test. This could be as simple as returning true because 3 is divisible by 3.

4. Implement the divisibleBy function to pass the first test:
   a. Add the following code to the `fizzbuzz.js` file above the line that starts with "`module.exports`":

```
FizzBuzz.prototype.divisibleBy = function(number, divisor) {
      return true;
};
```

b. Save the file. You should now see all your tests pass:

```
Fizzbuzz
    divisibleBy()
      ✓when divisible

  jshint
    ✓should pass for working directory (61ms)


  2 passing (67ms)
```

This code is clearly not correct, but it does pass the test. Another test is needed to test when the divisibleBy function should return false. Having the additional test will also require the correct implementation of the divisibleBy function.

5. Write the test and implement the code when divisibleBy should return false:
   a. In the `fizzbuzz.test.js` file, add code beneath the existing test to test whether 2 is divisible by three. The section of the test for the divisibleBy function should now look like this:

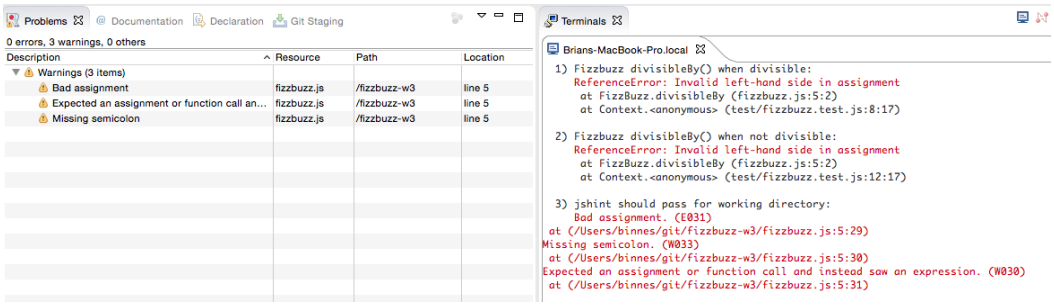
```
describe("divisibleBy()", function() {
  it("when divisible", function() {
    expect(f.divisibleBy(3, 3)).to.be.eql(true);
  });

  it("when not divisible", function() {
    expect(f.divisibleBy(3, 2)).to.be.eql(false);
  });
});
```

   b. Save the file. Now, you should have a failing test again.

   c. Implement the divideBy function with the following code. There is a deliberate coding error, so copy the code below as shown:

```
FizzBuzz.prototype.divisibleBy = function(number, divisor) {
      return number % divisor = 0;
};
```

Three errors are reported. You've broken the divideBy test for 3 divided by 3: the new test doesn't pass, and you also have JSHint complaining of bad JavaScript code. Notice that Eclipse is also reporting the JSHint issues in the problem panel.



d. Replace the code with the following new code. There is still a deliberate mistake.

```
FizzBuzz.prototype.divisibleBy = function(number, divisor) {
      return number % divisor == 0;
};
```

The functional tests pass, but JSHint is still complaining of bad JavaScript code. This is because in JavaScript there are two comparison operators:

- == does type coercion
- === does *not* do type coercion

Type coercion occurs when a variable is automatically converted to a different type when required, such as changing a number to a string.

e. When you use a static number, use the === operator. Replace the code again to use this code:

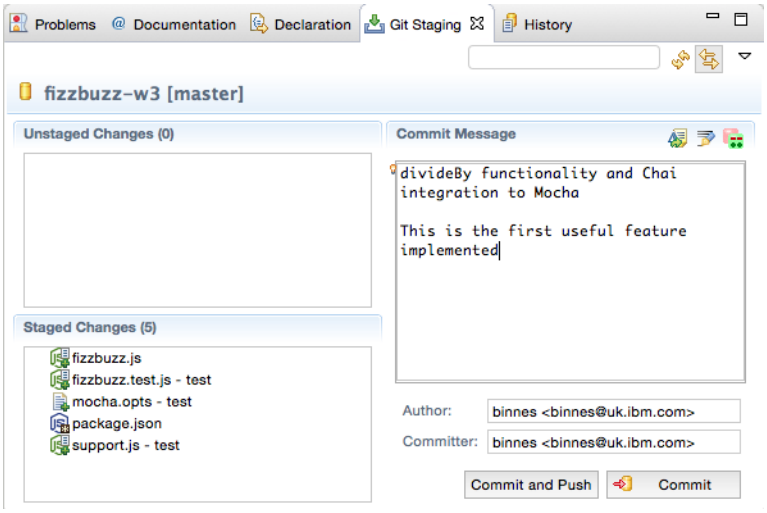
```
FizzBuzz.prototype.divisibleBy = function(number, divisor) {
      return number % divisor === 0;
};
```

All the tests now pass.

The first function is now implemented and all the tests pass. This is a good time to commit code:

6. Commit the code and push to the master repo:
   a. In the Git Staging view, select the files in the Unstaged Changes window and drag to the Staged Changes window. Add a comment and then click **Commit and Push**.

      Tip: It's better to split the configuration of Chai and Mocha into separate commits.



Implement the convertToFizzBuzz function

Now that you have a function to check whether a number is divisible by 3 or 5, you must implement a function to convert a number to its FizzBuzz value.

1. Implement the first test for convertToFizzBuzzZ:
   a. In the `fizzbuzz.test.js` file, add the following new test block after the divisibleBy block:

```
describe("divisibleBy()", function() {
  it("when divisible", function() {
    expect(f.divisibleBy(3, 3)).to.be.eql(true);
  });

  it("when not divisible", function() {
    expect(f.divisibleBy(3, 2)).to.be.eql(false);
  });
});

   describe("convertToFizzBuzz()", function() {
   });
```

   b. Add the first test to check that 3 is converted to “Fizz”:

```
describe("convertToFizzBuzz()", function() {
    it("when divisible by 3", function() {
     expect(f.convertToFizzBuzz(3)).to.be.eql("Fizz");
    });
  });
```

   c. In the `fizzbuzz.js` file, add a new function prototype for convertToFizzBuzz and add the code that is required to pass the first test:

```
FizzBuzz.prototype.convertToFizzBuzz = function(number) {
    return "Fizz";
};
```

   d. Add another test for Fizz to test that 6 also returns “Fizz” and then add tests to check that 5 and 10 return “Buzz” by updating the test code in the `fizzbuzz.test.js` file:

```
describe("convertToFizzBuzz()", function() {
    it("when divisible by 3", function() {
      expect(f.convertToFizzBuzz(3)).to.be.eql("Fizz");
      expect(f.convertToFizzBuzz(6)).to.be.eql("Fizz");
    });

    it("when divisible by 5", function() {
      expect(f.convertToFizzBuzz(5)).to.be.eql("Buzz");
      expect(f.convertToFizzBuzz(10)).to.be.eql("Buzz");
    });
});
```

   e. Implement the functionality in the `fizzbuzz.js` file to pass the tests by using the following code:

```
FizzBuzz.prototype.convertToFizzBuzz = function(number) {
 if (this.divisibleBy(number, 3)) {
    return "Fizz";
  }

  if (this.divisibleBy(number, 5)) {
    return "Buzz";
  }
};
```

All the tests should now pass.

f. Add tests to check for “FizzBuzz” and a number when that number is not divisible by either 3 or 5. Use the following code for the tests:

```
describe("convertToFizzBuzz()", function() {
  it("when divisible by 3", function() {
    expect(f.convertToFizzBuzz(3)).to.be.eql("Fizz");
    expect(f.convertToFizzBuzz(6)).to.be.eql("Fizz");
  });

  it("when divisible by 5", function() {
    expect(f.convertToFizzBuzz(5)).to.be.eql("Buzz");
    expect(f.convertToFizzBuzz(10)).to.be.eql("Buzz");
  });

  it("when divisible by 15", function() {
    expect(f.convertToFizzBuzz(15)).to.be.eql("FizzBuzz");
    expect(f.convertToFizzBuzz(30)).to.be.eql("FizzBuzz");
  });

  it("when not divisible by 3, 5 or 15", function() {
    expect(f.convertToFizzBuzz(4)).to.be.eql("4");
     expect(f.convertToFizzBuzz(7)).to.be.eql("7");

  });
});
```

g. To get the tests to pass, complete the convertToFizzBuzz function. Use the following code:

```
FizzBuzz.prototype.convertToFizzBuzz = function(number) {
  if (this.divisibleBy(number, 15)) {
    return "FizzBuzz";
  }

  if (this.divisibleBy(number, 3)) {
    return "Fizz";
  }

  if (this.divisibleBy(number, 5)) {
    return "Buzz";
  }

  return number.toString();
};
```

All the tests should now pass, and the function is complete.

h. Commit the code. Use the Git Staging view to stage, add a comment, and then commit and push the changes to the master repo.

Implement convertRangeToFizzBuzz (introduces Sinon.JS)

The next stage to implementing FizzBuzz is to be able to convert a range of numbers, not just a single number. Ensure that the tests for this function do not repeat the tests for convertToFizzBuzz. You already have tests for that.

The tests need to:

- Verify that when a range is converted, the results are returned in the correct order
- Verify that for a range, you call the convertToFizzBuzz function once for each member of the array

To enable this type of testing, an additional testing capability is needed. You'll use Sinon.JS to provide this capability. Sinon.JS provides a library to help you unit test your code. It supports spies, stubs, and mocks. The library supports multiple browsers and can run on a server using Node.js.

1. Add Sinon.JS and configure the test framework to use it:
   a. In the Terminals view, enter Ctrl+C to stop the Mocha tests. On Windows, answer Y to terminate batch job.

   b. Enter the following command in the Terminals window:

      ```
      npm install sinon --save-dev
      npm install sinon-chai --save-dev
      ```

   c. Refresh the Eclipse project by selecting the project name in the Project Explorer view and pressing F5.

   d. In the Terminals view, restart the Mocha `-w` command.

      Tip: Use the up arrow to scroll through previous commands.

      Mac and Linux: `node_modules/.bin/mocha -w`
      Windows: `node_modules\.bin\mocha -w`

   e. Modify the `support.js` file in the `test` folder to enable Chai to use Sinon.JS:

      ```
      var chai = require("chai");
      var sinonChai = require("sinon-chai");

      chai.use(sinonChai);
      global.expect = chai.expect;
      ```

2. Add the test for the convertRangeToFizzBuzz function:
   a. Add the Describe function below that tests for the convertToFizzBuzzz function:

      ```
        describe("convertRangeToFizzBuzz()", function() {
      });
      ```

   b. Add a test to ensure that the results are returned in the correct order:

```
  describe("convertRangeToFizzBuzz()", function() {
    it("returns in correct order", function() {
      expect(f.convertRangeToFizzBuzz(1, 3)).to.be.eql(["1", "2",
"Fizz"]);
    });

 });
```

c. Implement the function to satisfy the test by adding the following code:

```
FizzBuzz.prototype.convertRangeToFizzBuzz = function(start, end) {
  return ["1", "2", "Fizz"];
};
```

d. Add a test to ensure that the convertRangeToFizzBuzz is called the correct number of times for a given range and is called once for each number in the range. To do this, you use Sinon.JS to *spy* on the invocation of the convertToFizzBuzz function. At the top of the `fizzbuzz.test.js` file, add the following line of code to make Sinon.JS available:

```
var sinon = require("sinon");
```

The code for the convertRangeToFizzBuzz tests is now:

```
  describe("convertRangeToFizzBuzz()", function() {
    it("returns in correct order", function() {
      expect(f.convertRangeToFizzBuzz(1, 3)).to.be.eql(["1", "2",
"Fizz"]);
    });

    it("applies FizzBuzz to every number in the range", function() {
      var spy = sinon.spy(f, "convertToFizzBuzz");

      f.convertRangeToFizzBuzz(1, 50);

      for (var i = 1; i <= 50; i++) {
        expect(spy.withArgs(i).calledOnce).to.be.eql(true, "Expected
convertToFizzBuzz to be called with " + i);
      }
      f.convertToFizzBuzz.restore();
    });
  });
```

e. Implement the function to pass the tests by adding the following code:

```
FizzBuzz.prototype.convertRangeToFizzBuzz = function(start, end) {
  var result = [];

  for (var i = start; i <= end; i++) {
    result.push(this.convertToFizzBuzz(i));
  }

  return result;
};
```

The implementation of FizzBuzz is now complete because the code passes all the tests.

f. Commit the code by using the Git Staging view to stage, provide a comment, and then commit and push the changes to the master repository.